Jeu à deux joueurs à somme nulle et à information complète
Application au puissance 4

1 Introduction

L'objectif de ce TP est de programmer un jeu à deux joueurs à somme nulle et à information complète. Chacun des deux joueurs pourra être au choix un humain ou l'ordinateur. Si le joueur est l'ordinateur, il utilisera un algorithme classique et générique pour ce type de jeu : le Min-Max. Cet algorithme peut être optimisé (par élagage de l'arbre d'exploration) dans une version appelée Alpha-Beta que nous n'étudierons pas ici. L'algorithme étudié ici s'inscrit dans le cadre général de la théorie des jeux, défini par Wikipédia comme :

> *"ensemble d'outils pour analyser les situations dans lesquelles ce qu'il est optimal de faire pour un agent (personne physique, entreprise, animal, ...) dépend des anticipations qu'il forme sur ce qu'un ou plusieurs autres agents vont faire. L'objectif de la théorie des jeux est de modéliser ces situations, de déterminer une stratégie optimale pour chacun des agents, de prédire l'équilibre du jeu et de trouver comment aboutir à une situation optimale. La théorie des jeux est très souvent utilisée en économie, en sciences politiques, en biologie ou encore en philosophie."*

Ces deux définitions s'appliquent aux jeux classiques comme les dames, les échecs, othello, morpion, puissance 4, etc.

1.1 Définitions

Jeu à somme nulle. En théorie des jeux, un jeu est dit à somme nulle si la somme des gains de tous les joueurs est égale à 0. Dans le cas du jeu de dame par exemple, si on considère que le gain est de 1 pour le gagnant, -1 pour le perdant et 0 s'il y a match nul, le jeu est bien à somme nulle. En d'autres termes, la perte d'un joueur implique le gain d'un autre. L'intérêt d'un joueur est strictement opposé à l'intérêt de son adversaire. Cela correspond en économie à des situations où il n'y a pas de destructions ou de créations de produits.

Information complète. Un jeu est dit à information complète si un joueur connaît à chaque instant le but de son ou ses adversaires, leurs possibilités d'action et les gains en résultant.

1.2 Algorithme Min-Max

Considérons un jeu simpliste de type Morpion à une dimension (jeu sans aucun intérêt si ce n'est d'illustrer l'algorithme). Le joueur 1 (l'ordinateur) pose des croix, le joueur 2 (l'adversaire) pose des ronds. Les participants jouent à tour de rôle. Le premier joueur qui aligne 3 de ses pions a gagné. Si plus aucun pion ne peut être posé, la partie est nulle. Considérons l'état illustré figure 1. L'ordinateur doit choisir son prochain coup. En

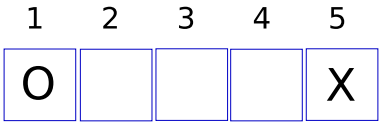

Figure 1: Exemple d'état d'un jeu de morpion à une dimension

considérant la règle du jeu, la liste des coups autorisés est $\{2, 3, 4\}$. Chacun de ces coups conduit aux états illustrés figure 2. Le choix du coup joué par l'ordinateur doit correspondre

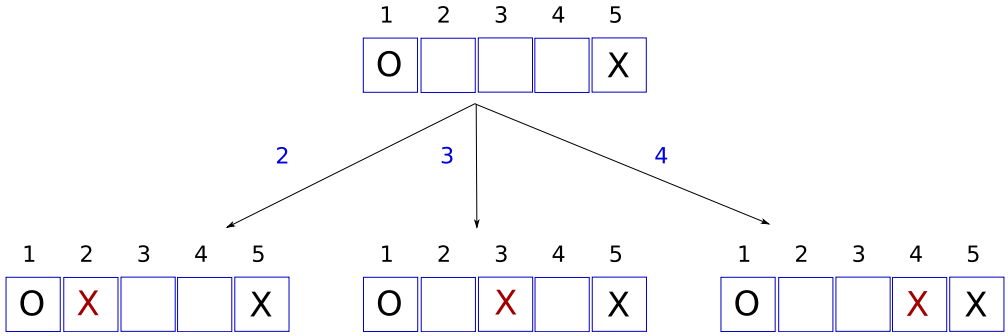

Figure 2: Etats obtenus à partir de l'ensemble des coups valides

à celui l'amenant dans l'état qui lui est le plus favorable. Un état est d'autant plus favorable qu'il est susceptible de conduire l'ordinateur à la victoire. Il est d'autant plus défavorable qu'il conduit l'adversaire à la victoire. Il faut donc imaginer l'ensemble des prochains coups possibles (jusqu'aux situations terminales) pour réaliser cette évaluation (cf figure 3). Dans le cas présent, les deux coups possibles pour l'ordinateur sont donc le 3 et le 4, les deux pouvant potentiellement conduire à une victoire.

Cette exploration de l'arbre de tous les coups possibles afin d'évaluer le prochain coup à jouer n'est possible que pour des jeux triviaux comme ce morpion à une dimension. Dans le cas de jeux plus intéressants comme les dames ou les échecs par exemple, une exploration systématique des coups possibles conduit à un arbre dont la taille explose. Dans le cas du jeu d'échec, le nombre total de situations différentes est de 10^{120}. Par comparaison, le nombre total d'atomes dans l'univers est estimé à 10^{80} .... Il faut donc mettre en place une *heuristique* pour déterminer le prochain coup à jouer, de manière pas nécessairement optimale mais en un temps réalisable.

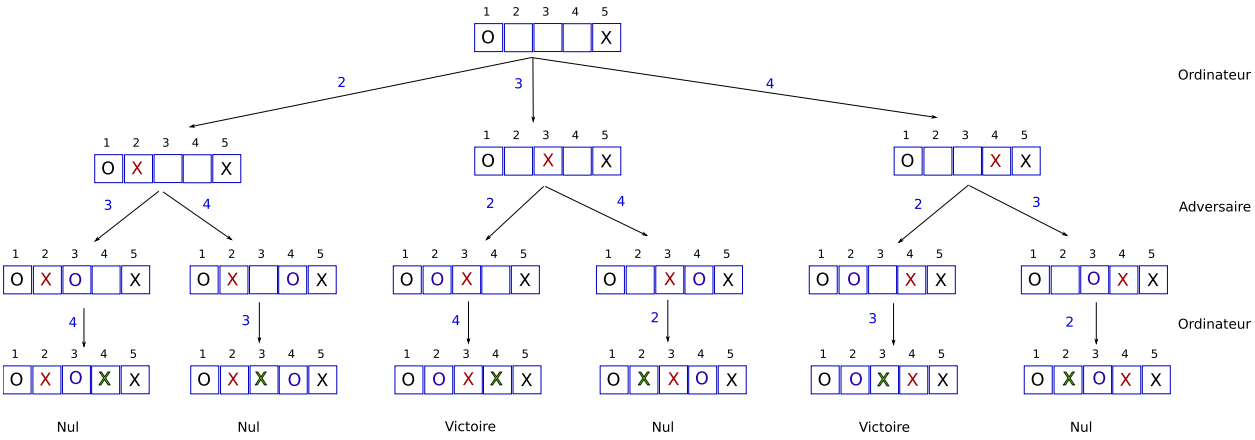

Figure 3: Ensemble des coups possibles jusqu'aux situations terminales

L'algorithme est le suivant. L'ordinateur doit choisir à partir de l'état reçu en paramètre son prochain coup à jouer. Il recherche l'ensemble des coups possibles depuis cet état et les nouveaux états en résultant. Il évalue le gain attendu pour chacun de ces états et choisi le coup conduisant vers l'état de gain maximum. L'évaluation du gain attendu est réalisé par l'algorithme MinMax.

L'algorithme MinMax reçoit en paramètre un état et une profondeur maximum de recherche et retourne le gain attendu pour cet état (sa valeur).

- Si la profondeur est nulle (on a atteint une feuille dans notre recherche, mais qui n'est pas forcément une situation terminale), on réalise une évaluation "statique" de l'état courant. Cela signifie qu'on ne cherche plus à se projeter sur les coups qui peuvent en découler mais on analyse l'état (la position respective des différents pions par exemple dans un jeu de plateau) pour fournir une appréciation sur la situation courante. Cette appréciation doit être un nombre positif d'autant plus grand que la situation est favorable à l'ordinateur, négatif et d'autant plus petit qu'elle est favorable à l'adversaire et nulle si la situation est neutre (on ne sait pas juger dans un sens ou dans l'autre).

- Si la profondeur est non nulle mais l'état est une situation terminale, sa valeur est $+v$ si l'ordinateur a gagné, $-v$ si l'adversaire a gagné, 0 si la situation est nulle. Remarque : la valeur retournée par l'évaluation statique (point précédent) ne doit pas excéder v.

- Dans tous les autres cas, la valeur de l'état courant est :

  – le minimum de la valeur des états atteignables en un coup si c'est à l'adversaire de jouer (on fait donc l'hypothèse que l'adversaire jouera un coup qui le favorise le plus et donc qui défavorise l'ordinateur le plus, le jeu étant à somme nulle).

– le maximum de la valeur des états atteignables en un coup si c'est à l'ordinateur de jouer (correspond à l 'hypothèse que l'ordinateur jouera le coup qui le favorise le plus).

Cet algorithme est illustré sur la figure 4.

Remarques

1. De manière simple, on peut analyser cet algorithme de la manière suivante. Choisir le prochain coût en analysant statiquement les états atteignables en 1 coup par l'ordinateur ne fonctionne pas car il est très difficile de fournir une fonction d'évaluation performante qui n'analyse que la situation courante. On compense donc la "faiblesse" de la fonction d'évaluation en n'analysant non pas à 1 coup en avance mais en envisageant une succession de coups joués alternativement par l'ordinateur et le joueur.

2. A profondeur de recherche égale, la fonction d'évaluation est ce qui fait la différence de niveau entre deux programmes.

3. Lors du choix du prochain coup à jouer par l'ordinateur, plusieurs états peuvent avoir le score maximum. Il faut alors **tirer aléatoirement** le vainqueur de manière à apporter un peu de variabilité dans le jeu.

2 Présentation des fichiers fournis

De manière à guider le développement, nous vous fournissons l'architecture suivante à travers la spécification de différents packages :

2.1 Participant

```
package Participant is

    type Joueur is (Joueur1, Joueur2);

    -- Retourne l'adversaire du joueur passe en parametre
    function Adversaire(J : Joueur) return Joueur;

end Participant;
```

Nous ne considérons que les jeux à deux joueurs, respectivement `Joueur1` et `Joueur2`

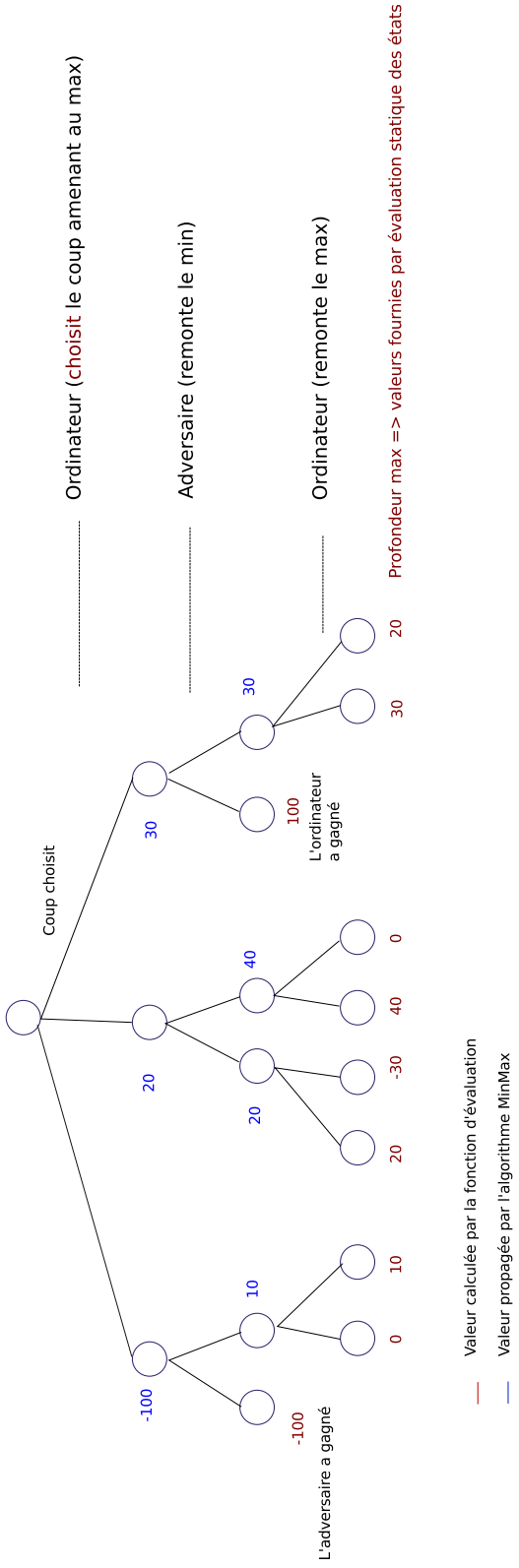

Figure 4: Exemple de choix d'un coup grâce à une évaluation MinMax

2.2 Partie

Quel que soit le jeu, une partie à deux joueurs est une série de coups joués successivement par les deux opposants jusqu'à obtenir un état correspondant à *gagne*, *perdu* ou *nul*. Ce déroulement de partie peut donc être abstrait dans un package générique dont les paramètres vont permettre de spécifier le jeu manipulé. Ce package générique est "l'interface homme-machine" de votre jeu.

```
generic

    type Etat is private;
    type Coup is private;

    -- Nom affichable du Joueur1
    Nom_Joueur1 : String;
    -- Nom affichable du Joueur2
    Nom_Joueur2 : String;
    -- Calcule l'etat suivant en appliquant le coup
    with function Etat_Suivant(E : Etat; C : Coup) return Etat;
    -- Indique si l'etat courant est gagnant pour le joueur J
    with function Est_Gagnant(E : Etat; J : Joueur) return Boolean;
    -- Indique si l'etat courant est un status quo (match nul)
    with function Est_Nul(E : Etat) return Boolean;
    -- Fonction d'affichage de l'etat courant du jeu
    with procedure Affiche_Jeu(E : Etat);
    -- Affiche a l'ecran le coup passe en parametre
    with procedure Affiche_Coup(C : in Coup);
    -- Retourne le prochaine coup joue par le joueur1
    with function Coup_Joueur1(E : Etat) return Coup;
    -- Retourne le prochaine coup joue par le joueur2
    with function Coup_Joueur2(E : Etat) return Coup;

package Partie is

    -- Joue une partie.
    -- E : Etat initial
    -- J : Joueur qui commence
    procedure Joue_Partie(E : in out Etat; J : in Joueur);

end Partie;
```

Un jeu est défini par son `Etat`, la description d'un `Coup` et le nom affichable des deux joueurs. La mécanique du jeu est spécifiée par la fonction `Etat_Suivant`. Les Etats terminaux sont détectés par `Est_Gagnant` et `Est_Nul`. L'affichage à l'écran est réalisé par les fonction `Affiche_Jeu` et `Affiche_Coup`. Les deux derniers paramètres sont les fonctions à la disposition du package `Partie` pour demander au `Joueur1` (respectivement `Joueur2`) le prochain coup qu'il souhaite jouer. Ces deux fonctions peuvent être une demande au clavier ou l'algorithme `MinMax`. Une partie peut donc être entre deux joueurs humains, un humain

et l'ordinateur ou l'ordinateur contre lui-même. Cette dernière option peut être intéressante pour vérifier si par exemple une recherche avec une profondeur plus importante gagne en moyenne plus souvent contre une profondeur moins importante.

2.3 Moteur_Jeu

Le moteur du jeu est une implémentation de l'algorithme `MinMax` permettant à partir d'un état donné de sélectionner le coup suivant. L'algorithme `MinMax` étant indépendant du jeu, le package est générique. Les paramètres permettant de spécifier le jeu sont les mêmes que ceux de `Partie` auxquels on rajoute la fonction `Eval` d'évaluation statique (évaluation de la situation courante lorsque la profondeur de recherche est atteinte) et la fonction permettant de fournir la liste de coups possibles pour un état donné. Cette fonction implémente les règles du jeu. Remarque : la fonction `Eval` se place du point de vue de l'ordinateur. Elle fournit une valeur positive élevée si la position évaluée est intéressante du point de vue de celui-ci, une valeur négative faible si elle est intéressante du point de vue de l'adversaire et nulle si neutre.

```
generic
    type Etat is private;
    type Coup is private;

    -- Calcule l'etat suivant en appliquant le coup
    with function Etat_Suivant(E : Etat; C : Coup) return Etat;
    -- Indique si l'etat courant est gagnant pour J
    with function Est_Gagnant(E : Etat; J : Joueur) return Boolean;
    -- Indique si l'etat courant est un status quo (match nul)
    with function Est_Nul(E : Etat) return Boolean;

    -- Affiche a l'ecran le coup passe en parametre
    with procedure Affiche_Coup(C : in Coup);
    -- Implantation d'un package de liste de coups
    with package Liste_Coups is new Liste_Generique(Coup, Affiche_Coup);
    -- Retourne la liste des coups possibles pour J a partir de l'etat
    with function Coups_Possibles(E : Etat; J : Joueur)
            return Liste_Coups.Liste;
    -- Evaluation statique du jeu du point de vue de l'ordinateur
    with function Eval(E : Etat) return Integer;
    -- Profondeur de recherche du coup
    P : Natural;
    -- Indique le joueur interprete par le moteur
    JoueurMoteur : Joueur;

package Moteur_Jeu is

    -- Choix du prochain coup par l'ordinateur.
    -- E : l'etat actuel du jeu;
    -- P : profondeur a laquelle la selection doit s'effetuer
```

```
    function Choix_Coup(E : Etat) return Coup;

private
    -- Evaluation d'un coup a partir d'un etat donne
    -- E : Etat courant
    -- P : profondeur a laquelle cette evaluation doit etre realisee
    -- C : Coup a evaluer
    -- J : Joueur qui realise le coup
    function Eval_Min_Max(E : Etat; P : Natural; C : Coup; J : Joueur)
        return Integer;

end Moteur_Jeu;
```

Remarque : la fonction `Coups_Possibles` renvoie une liste de coups. Le type `Coup` est un des paramètres de la généricité. Le type "liste de coups" doit donc également être un paramètre de cette généricité. On aurait pu l'ajouter comme paramètre supplémentaire :

```
generic
    type Etat is private;
    type Coup is private;
    type Liste_De_Coups is private;
    .....
    with function Coups_Possibles(E : Etat; J : Joueur)
            return Liste_De_Coups;
```

En procédant de cette façon, `Coup` et `Liste_De_Coups` auraient alors été deux types sans aucune relation particulière entre eux. Il aurait fallu rajouter en paramètres supplémentaires de la généricité l'ensemble des fonctions de manipulation de listes, recréant ce lien entre ces deux types :

```
generic
    type Etat is private;
    type Coup is private;
    type Liste_De_Coups is private;
    .....
    with procedure Insere_Tete (V : in Coup; L : in out Liste_De_Coups);
    with function Creer_Iterateur (L : Liste_De_Coups) return Iterateur;
    with function Coup_Courant(It : Iterateur) return Coup;
    .....
    with function Coups_Possibles(E : Etat; J : Joueur)
            return Liste_De_Coups;
```

L'autre solution est d'utiliser un type de liste générique (cf section 2.4) et de définir au niveau des paramètres le package de liste de coups comme une instance du package de liste générique avec `Coup` comme type d'objet manipulé.

```
with Liste_Generique;

generic
    type Etat is private;
```

```
    type Coup is private;

    with package Liste_Coups is new Liste_Generique(Coup, Affiche_Coup);
    .....
```

L'ensemble des fonctions et procédures de manipulation de ces listes est apporté directement par le package de liste.

2.4 Liste_Generique

Le package de liste chainée générique est défini par :

```
generic
    type Element is private;
    with procedure Put(E : in Element);

-- Les specifications du package, qui n'utilisent
-- que les elements et procedures generiques
package Liste_Generique is
    type Liste is private;
    type Iterateur is private;

    -- Affichage de la liste, dans l'ordre de parcours
    procedure Affiche_Liste (L : in Liste);

    -- Insertion d'un element V en tete de liste
    procedure Insere_Tete (V : in Element; L : in out Liste);

    -- Vide la liste et libere toute la memoire utilisee
    procedure Libere_Liste(L : in out Liste);

    -- Creation de la liste vide
    function Creer_Liste return Liste;

    -- Cree un nouvel iterateur
    function Creer_Iterateur (L : Liste) return Iterateur;

    -- Liberation d'un iterateur
    procedure Libere_Iterateur(It : in out Iterateur);

    -- Avance d'une case dans la liste
    procedure Suivant(It : in out Iterateur);

    -- Retourne l'element courant
    function Element_Courant(It : Iterateur) return Element;

    -- Verifie s'il reste un element a parcourir
    function A_Suivant(It : Iterateur) return Boolean;
```

```
    FinDeListe : exception;

private
    type Cellule;
    type Liste is access Cellule;

    type Iterateur_Interne;
    type Iterateur is access Iterateur_Interne;

end Liste_Generique;
```

Le parcours de la liste et l'accès aux différents éléments n'est possible qu'à travers un itérateur. Ce mécanisme est similaire à celui fourni par Java.

3 Le puissance 4

Le puissance 4 est un jeu dont le plateau est un ensemble de colonnes. Chaque joueur possède son type de pion. A chaque tour, le joueur choisit la colonne dans laquelle mettre son pion. Celui-ci ira se positionner dans la case vide la plus basse. Si la colonne est pleine, le pion ne peut pas y être déposé. Le gagnant est le premier à aligner un nombre de pions fixé au départ. Cet alignement peut être vertical, horizontal ou diagonal. Si toutes les cases sont pleines sans alignement, la partie est nulle.

On souhaite dans le cadre de ce projet réaliser un package de Jeu `Puissance4` générique dont les paramètres de la généricité seront la largeur du plateau, sa hauteur et le nombre de pions alignés nécessaires pour la victoire.

Exemple de déroulement du jeu :

```
$ ./main2joueurs
Puissance 4

Joueur 1 : X
Joueur 2 : O

--------------------------------------------------------------------------------

  1  2  3
|  |  |  |
|  |  |  |
|  |  |  |
----------
Paul : Numéro de Colonne : 1
Joueur 2 joue :  1
```

```
--------------------------------------------------------------------------------

  1  2  3
|  |  |  |
|  |  |  |
|O |  |  |
----------
Pierre : Numéro de Colonne : 3
Joueur 1 joue :  3

--------------------------------------------------------------------------------

  1  2  3
|  |  |  |
|  |  |  |
|O |  |X |
----------
Paul : Numéro de Colonne : 3
Joueur 2 joue :  3

--------------------------------------------------------------------------------

  1  2  3
|  |  |  |
|  |  |O |
|O |  |X |
----------
Pierre : Numéro de Colonne : 2
Joueur 1 joue :  2

--------------------------------------------------------------------------------

  1  2  3
|  |  |  |
|  |  |O |
|O |X |X |
----------
Paul : Numéro de Colonne : 2
Joueur 2 joue :  2

--------------------------------------------------------------------------------

  1  2  3
|  |  |  |
```

```
|  |O |O |
|O |X |X |
----------
Pierre : Numéro de Colonne : 1
Joueur 1 joue :  1


--------------------------------------------------------------------------------

  1  2  3
|  |  |  |
|X |O |O |
|O |X |X |
----------
Paul : Numéro de Colonne : 3
Joueur 2 joue :  3
  1  2  3
|  |  |O |
|X |O |O |
|O |X |X |
----------
Paul a gagné
```

4 Objectifs et travail à rendre

L'objectif est de fournir une implémentation des différents packages fournis (`Liste_Generique`, `Partie` et `Moteur_Jeu` et de les appliquer à un jeu de Puissance 4.

Pour simplifier le développement, nous vous conseillons dans un premier temps de constuire le package `Partie` et `Puissance4` sans la fonction d'évaluation ni la liste de coups. Vous pourrez ainsi les tester dans le cadre d'une partie entre deux joueurs humains. Construisez ensuite le moteur de jeu de manière à remplacer au moins un des joueurs humain par l'ordinateur. Le moteur de jeu et de parties étant générique, vous pouvez implémenter un autre jeu si le cœur (et le temps...) vous en dit.

Le TP est à réaliser en binôme. Il devra être rendu sur Teide et devra comporter :

- l'ensemble de vos fichiers sources lisibles et commentés.
- un court rapport (4 pages maximum) pour expliquer et justifier vos structures de données et principaux algorithmes, leurs coûts et présenter les résultats obtenus.

La date de rendu du TP est fixée au **vendredi 8 décembre 2017**